

Beam-correlated Data Acquisition

15Hz beam data access

Fri, May 19, 2000

The current Acnet console model for data pool management does not yield reliable access to 15Hz data that is correlated across multiple front ends. With increased attention paid to Booster studies and the increased complexity of multiple accelerator operational modes, the need for such data is expected to become more important than before. This note describes one method for supporting access to correlated data with suitable cooperation between a console application and front end software.

Acnet console data delivery system

The interface between the Acnet console system and a user application is via a data pool that is maintained by the console system. Reply data coming from front ends, in response to data requests made by various applications running on the console computer, are copied into the data pool when they arrive. The application program executes at an approximate 15Hz rate and can ask for the latest data pool values at any time. But it is an asynchronous polling scheme that cannot reliably collect data from each 15Hz cycle, even if the replies arrive from front ends at 15Hz. The 15Hz execution of console applications is based upon timing services provided by the console operating system, when supports delays in units of 10 ms. In order to achieve an approximate 15Hz execution model, delays of 70, 70, and 60 ms are used for scheduling successive executions of an application program.

If changes were made to this console scheduling paradigm, say to provide callbacks that execute whenever new data values arrive from the front ends, the situation may improve. But with the need for allocating scarce programmer resources toward the new Java-based Acnet system, no substantive console changes are anticipated. This note assumes that the current Acnet execution model will persist for the near future. Any means of providing access to correlated 15Hz data must be based upon what exists.

Polling for 15Hz data

One problem is how to arrange to use the present polling method to collect 15Hz data, when the polling occurs at 15Hz asynchronous to data arrival. The key is to have replies delivered by the front ends at slower rates than 15Hz, where the replies includes data measured from multiple 15Hz cycles. If replies are delivered at 7.5Hz, say, the 15Hz polling method is good enough to catch all such replies. The data pool has only one copy of a reply for a given Acnet device, so how can this work? The application must interpret the reply as a multiple-valued structure, breaking it down into its component parts for analysis.

Correlation across front ends

If all correlated data desired by an application is delivered to the console within one reply message from a single front end, correlation is easy. The call that asks for the entire set of data to be sampled from the data pool returns an error status if any of the data components were changed by arrival of a new reply message during the time that the data values in the set were being collected for delivery to the caller. In the event that such status is returned, the application merely has to repeat the call.

But it is quite likely that multiple front ends source the data that the application requires, so the above simplification does not apply. In order to determine which replies from which front ends belong together--measure the same 15Hz cycle beam--a time stamp is needed to tag each data item. A suitable time stamp for 15Hz accelerator data measurements is a 15Hz cycle counter that follows the accelerator cycles. In order to use a cycle counter to identify on which 15Hz cycle the data was measured, all front ends must know the same cycle counter value for any given accelerator cycle.

Front end 15Hz cycle counter

About 3/4 of the way through each 15Hz accelerator cycle, which is measured here from one

Booster BMIN to the next BMIN, the Tevatron clock event 0F occurs. In response to that, a special front end multicasts an ethernet datagram on UDP port number 50090 that includes a 15Hz cycle counter as well as information about recent clock events. The clock event information is used by some front ends that have no connection to the accelerator clock. The cycle counter may be used to establish a globally-recognized cycle counter for time stamp purposes.

A front end that measures 15Hz data will do so in synchronism with the accelerator clock, so that it knows how to time stamp the data it reads into its own data pool. Let us assume that the cycle counter that in the multicast datagram is for use on data that is measured on the following 15Hz cycle. There is an implicit assumption here that UDP datagrams are delivered promptly by the network hardware. This may not always be the case. Sometimes the arrival of the datagram may be delayed by several milliseconds, maybe even enough for it to arrive in the next 15Hz cycle. So it may be helpful to be aware of this possibility and adjust for it.

A 15Hz front end can count accelerator cycles. The only problem is to get in synchronism with the globally-accepted counter value, which is delivered by the multicast message. Suppose a front end listens to the multicast message, and if it arrives at approximately the expected time, it accepts it for use on the following cycle. If it does not arrive at the expected time, it is ignored, and a substitute value of the cycle counter is used, a value that is one more than the value used during the previous cycle. During normal accelerator operation, then, receiving one proper multicast message is enough; each subsequent cycle increments this value by one.

Take the case of a Booster IRM front end, whose 15Hz cycle operation starts at, say, 33 ms after BMIN. It knows when clock event 0F occurs, since it receives an interrupt for every clock event. When the multicast message is received, the local application called ACLK receives it, and it can save the cycle counter value. Let the value received be stored in a variable called *CycNew*, say, and the time of its arrival stored in *CycTim*. At the beginning of the node's 15Hz activity, when its data pool is updated, a check is made for the value of *CycTim* occurring within 10 ms from the time of the most recent 0F event. If it has, then it is accepted as the new value of *CycCnt*. If it has not then *CycCnt* is merely incremented by one. In either case, the value of *CycCnt* is associated with each data value in the updated data pool in the current cycle. *CycCnt* should only be modified when a node is updating the data pool.

Console application

What can the application do in order to interpret the data it receives at 7.5Hz, or slower? It must know a cycle counter value associated with each data item in the replies which it gleans from the console data pool. The cycle counter must come from the front end along with the data. In order to make use of true correlated data, the application must correlated readings of multiple devices by matching cycle counter values. This may seem complicated, but there is no obvious alternative given the application interface to the data pool.

Time-stamped data values in replies

If we have to deliver time-stamped data in response to console requests, what form will it take? The front end must accumulate multiple copies of data items to be time stamped, because they must be delivered at a slower rate than 15Hz. One approach to provide such replies is to use data streams, which is a formalized scheme for managing circular buffers in IRMs. Every cycle, in addition to updating the 15Hz data pool, the front end would also write into separate data streams defined for holding multiple copies of the measured data values.

Let there be a separate data stream defined for each data pool item. A 4-byte structure may be written

into each data stream that includes a 2-byte cycle counter and a 2-byte data value. When a 7.5Hz request becomes due, a 4-byte header and a 4-byte data-time structure would be returned for each such device. The header includes a count of the number of data values being returned and the size of one data-time value, say, 4. The Acnet console system will only store into its data pool the entire 8-byte structure, leaving the application to sort out when is contained therein and update its own data pool of recently-received data values and time stamps.

Is there an approach that does not use data streams that might be more efficient? The reply data furnished by the data stream method is probably necessary—except for the record size word—for the application; it needs to know how many data items in the structure are valid. (Maybe not; why should the number of valid structures vary? One reason is that the immediate reply may be empty, whereas the following replies may be full.)

Suppose there is a table in the front end that is used to house multiple copies of the data pool values. Indexing in this table should be the same for all devices measured in the same cycle. When building a reply value for a given device, the values from this table can be read out, and suitable cycle numbers attached to each. The result would be the same as data stream support provides, but it should be faster. A variation of this approach is to use a 2-word header to the data values. Let the first word indicate how many items are value and the second word indicate the cycle number of the first value. The application will know that the subsequent data item(s) use successive cycle numbers. (Of course, the application must know the size of each data item.)

As for the internal details of how this new listype code would work, the internal pointer value would not be a simple pointer to the data item in the data pool, but it would need to indicate which channel and which entry) was last delivered, so it can know where to find data items to use for the next reply. It can be initialized to point to the oldest data item in the buffer consistent with the size of the reply buffer.

Note that the scheme described in this note assumes that the application wants to collect 15Hz data; it does not allow the console to collect 5Hz data, for example. The console has no way to request 5Hz data under this scheme. If it makes a 5Hz request, it will receive 15Hz data items to fit in its buffer. If the buffer is too small, it will soon become too far behind. An event-based request will not work, either, because the events may occur at 15Hz, which will result in 15Hz replies, or the events may occur too slowly for the size of the the multiple-valued buffer in the front end. The request is for 15Hz data. In order to collect any event-based data, a device that delivers the clock event number must be correlated with the signal data.

A different scheme that would allow replies that include slower sampling and event-related data would mean that the front end would have to build up reply data for such a request at a rate that is slower than the reply rate. But the RETDAT protocol does not support this. Such an approach was included in the design of the D0 protocol that was used by the D0 detector during Run I.

If the period of the request is 1 second, and the buffer has room for 5 values, can we return every third value? This is a bad idea, because different front ends may return data from different cycles. The scheme is really designed for 15Hz access. The above case should return only the most recent 5 values, skipping 10 for every reply. This is probably not what is desirable. Note that one can receive uncorrelated, but time stamped, data at a slow rate by providing a buffer size only large enough for one item. The minimum buffer size must be 6 bytes, or two words. This would provide a count of 1, a cycle counter, and a single data item.